Load Balancer (LB)

Suresh Podeti
7 min read · Sep 20, 2023

Introduction

To serve millions of requests per second thousands of servers work together to share the load of incoming requests, is called **load balancing.** Incoming requests will be divided among all the available servers using a **load balancer**.

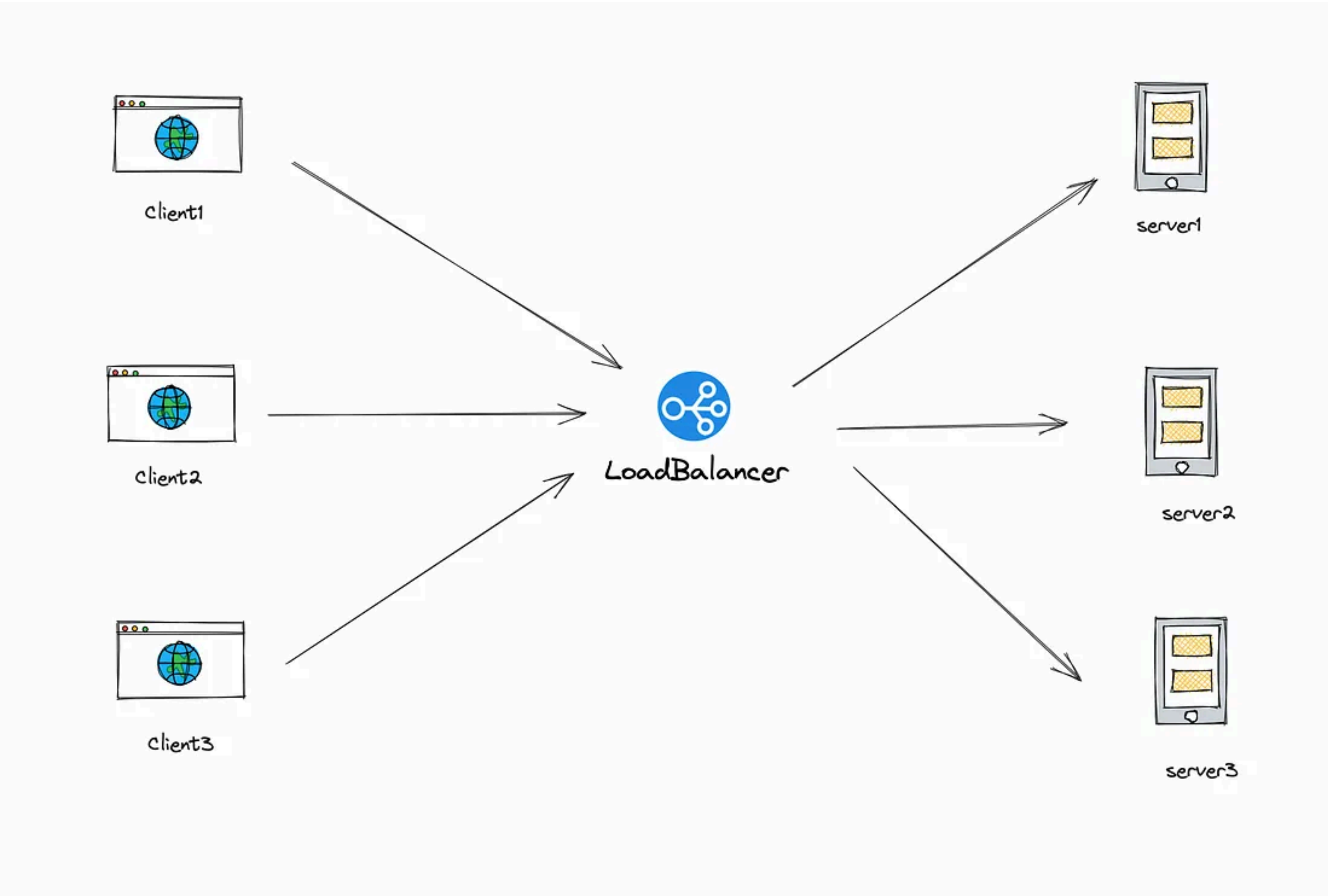

Fig 1.0: Simplified working of a load balancer

Why do we need to balance the load?

If we don't balance the load, all the load ends up on the same server leading server to overload and crash.

When do we not need the load balancer?

We may not be required, If the service entertains a few hundred or even few thousand request per second.

What are the guarantees provided by the load balancer?

Scalability — Capacity of the application/service can be increased seamlessly by adding more servers.

Availability — Even if some of the servers goes down, the system still remains available.

Performance — Load balancers distribute load to improves the systems performance as an overall.

Load balancers placement

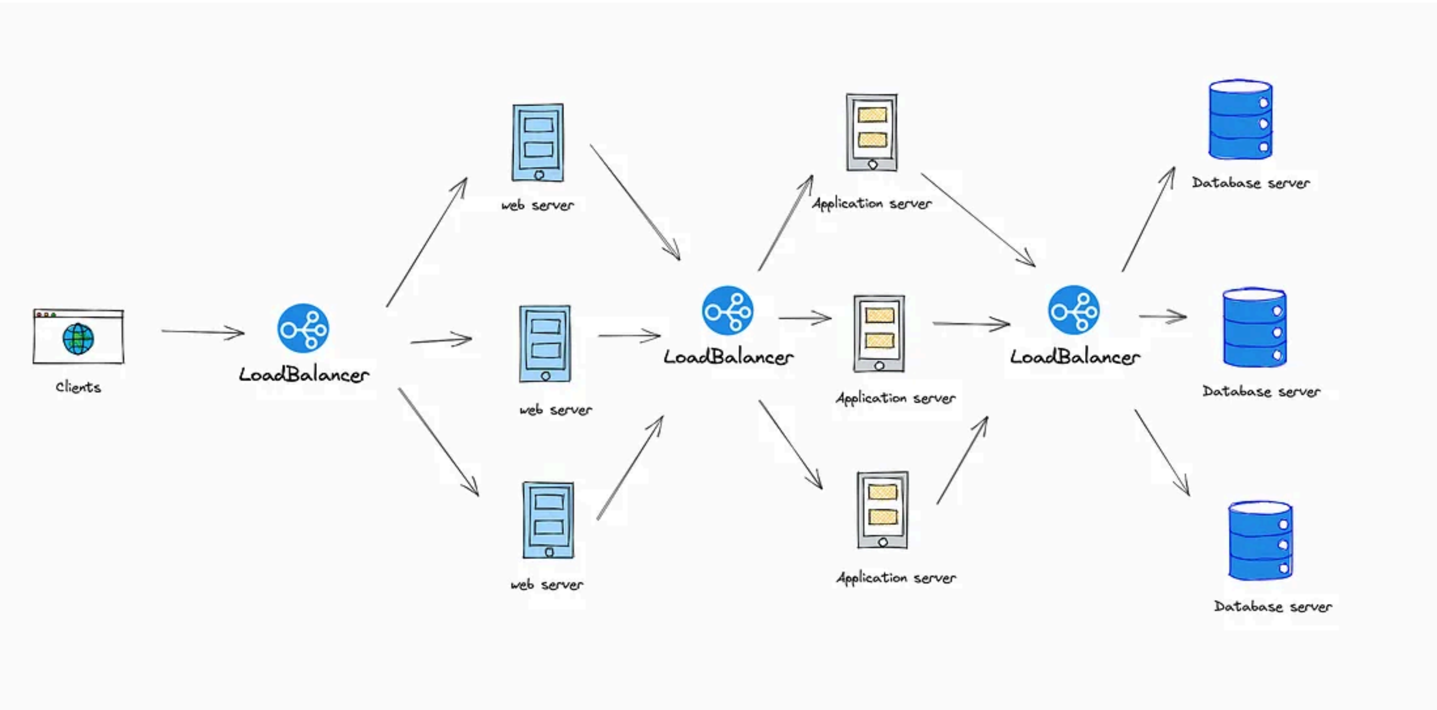

Fig 2.0: Possible usage of load balancers in a three-tier architecture

Load balancer can be placed potentially between any two services with multiple instances within the design of the system.

Load balancer other services

They offer some key services like **health checking, predictive analytics, reduced human intervention, TLS termination, and security**.

TLS termination —TLS termination is called "termination" because it refers to the termination or ending of the TLS (Transport Layer Security) or SSL (Secure Sockets Layer) encryption process at a specific network device, such as a load balancer. In this context, "termination" means the removal or termination of the encryption layer applied to network traffic.

TLS/SSL encryption and decryption can be computationally intensive, especially when dealing with a large volume of secure connections. By offloading this encryption/decryption process to a dedicated load balancer, the backend servers can focus on handling application-specific tasks, which can significantly improve their performance and response times.

Load balancing is required at two levels:

Global Server Load balancing (GSLB)

GSLB distribute incoming traffic to **nearest** based on the user's geographical location. Helps **minimise latency, improve overall user experience.** By leveraging GSLB, **CDN**s can deliver content from the nearest edge server to end-users, reducing latency and improving content delivery speed.

GSLB distributes load across multiple geographical regions based geographical locations, no.of hosting services in different locations, health of the datacenter.

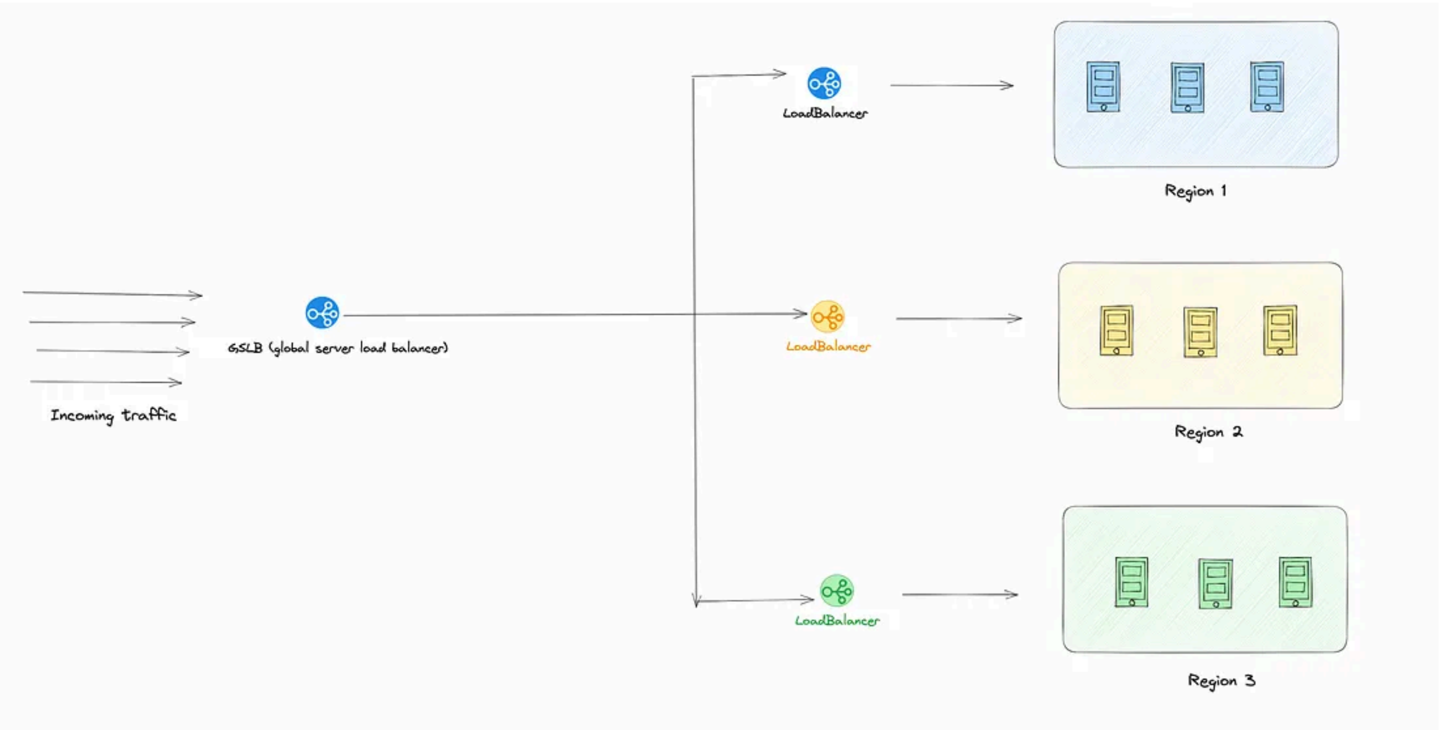

Fig 3.0: Usage of global load balancing to send user requests to different regions

Load balancing in DNS

DNS can respond with multiple IP addresses for a DNS query. It's possible to do load balancing through DNS while looking at the output of `nslookup` . DNS distributes the load of requests on different datacenters. This is performing **GSLB.** In particular, DNS uses **round-robin** to perform load balancing.

Run command: `dig google.com`

```
;; AUTHORITY SECTION:
google.com.  74550 IN NS ns4.google.com.
google.com.  74550 IN NS ns3.google.com.
google.com.  74550 IN NS ns1.google.com.
google.com.  74550 IN NS ns2.google.com.
```

```
;; AUTHORITY SECTION:
google.com.  74537 IN NS ns3.google.com.
google.com.  74537 IN NS ns1.google.com.
google.com.  74537 IN NS ns2.google.com.
google.com.  74537 IN NS ns4.google.com.
```

List of host names are reordered in the returned list as shown above if we run multiple times.

Local Load Balancing (LLB)

Why we need local load balancer although we have GSLB?

- The size of the DNS packet (512 bytes) isn't enough to include all the possible IP addresses of the servers.
- Clients can't determine the closest address to establish connection with.
- Some of the received IP address may belongs to busy data centers.

Resides within the datacenter, behaves like **reverse proxy** and make their best effort to divide incoming requests among the pool of available servers. Example: NGINX reverse proxy.

Algorithms

- **Round-robin scheduling —** Each request is forwarded to a server in the pool in a repeating sequential manner.
- **Weighted round-robin —** Same as round robin but prioritising the server with more capacity.
- **Least connections** — Arriving requests are assigned to servers with fewer existing connections.
- **Least response time** — Server with least response time is selected to server the request
- **IP hash** — Hashing the IP address is performed to assign users' requests to server.
- **URL hash**

Static vs dynamic algorithms

Static — Algorithms that don't consider **changing the state of servers.** Therefore, task assignment is carried out based on existing knowledge about the server's configuration.

Dynamic — Algorithms that consider the **current or recent state of the servers.** Dynamic algorithms maintain state by communicating with the server, which adds a communication overhead.

Stateful versus stateless LBs

Stateful — Involves maintaining a state of the sessions established between clients and hosting servers.

Stateless — Stateless load balancing maintains no state and is, therefore, faster and lightweight.

State maintained across different load balancers is considered as stateful load balancing. Whereas, a state maintained within a load balancer for internal use is assumed as stateless load balancing.

Types of load balancers

Load balancing can be performed at the **network/transport** and **application layer** of the open systems interconnection (OSI) layers.

Layer 4 load balancers — Load balancing performed on the basis of transport protocols like TCP and UDP. These types of LBs maintain connection/session with the clients and ensure that the same (TCP/UDP) communication ends up being forwarded to the same back-end server.

Layer 7 load balancers — Load balancers are based on the data of application layer protocols. It's possible to make application-aware forwarding decisions based on HTTP headers, URLs, cookies, and other application-specific data

Coding

Round robin algorithm

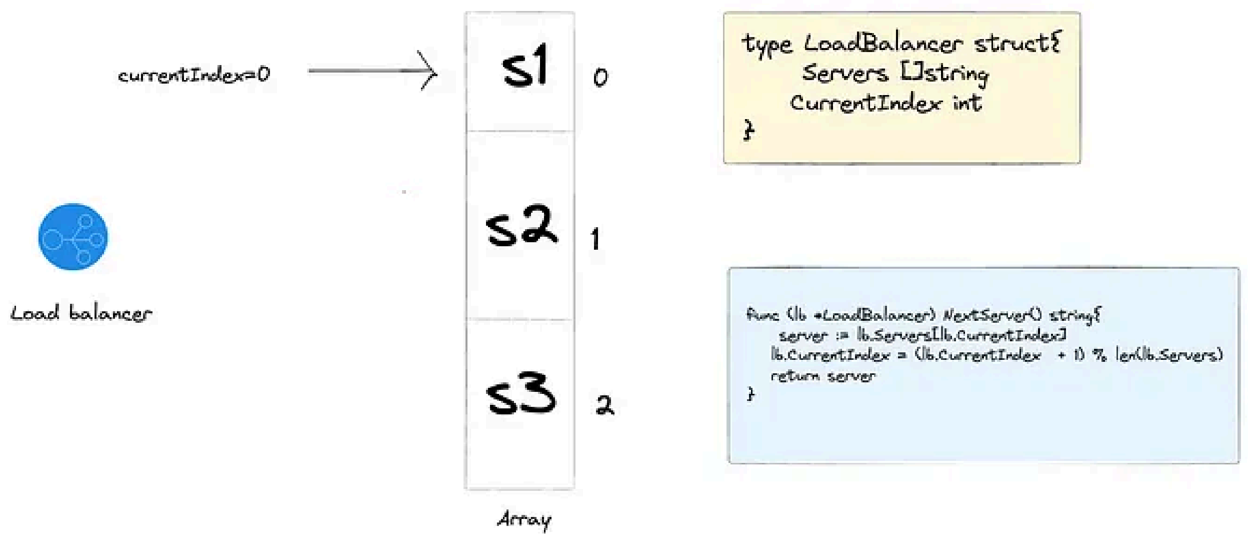

Fig 4.0: Round robin algorithm illustration

```
package main

import "fmt"

type LoadBalancer struct {
 Servers      []string
 CurrentIndex int
}

func (lb *LoadBalancer) NextServer() string {
 server := lb.Servers[lb.CurrentIndex]
 lb.CurrentIndex = (lb.CurrentIndex + 1) % len(lb.Servers)
 return server
}

func main() {
```

```
 servers := []string{"Server 1", "Server 2", "Server 3"}

 lb := &LoadBalancer{Servers: servers, CurrentIndex: 0}

 for i := 0; i < 10; i++ {
  server := lb.NextServer()
  fmt.Printf("server: %s\n", server)
 }

}
```

Weighted round robin

```
package main

import (
 "fmt"
)

type Server struct {
 Name   string
 Weight int
}

type LoadBalancer struct {
 servers   []Server
 currIndex int
 gcd       int
 maxWeight int
}

func NewLoadBalancer(servers []Server) *LoadBalancer {
 lb := &LoadBalancer{
  servers:   servers,
  currIndex: 0,
  gcd:       0,
  maxWeight: 0,
 }

 // Calculate the greatest common divisor (GCD) of server weights
 for _, server := range lb.servers {
  lb.gcd = gcd(lb.gcd, server.Weight)
  if server.Weight > lb.maxWeight {
   lb.maxWeight = server.Weight
  }
 }

 return lb
}
```

```
func gcd(a, b int) int {
 if b == 0 {
  return a
 }
 return gcd(b, a%b)
}

func (lb *LoadBalancer) NextServer() string {
 for {
  lb.currIndex = (lb.currIndex + 1) % len(lb.servers)
  if lb.currIndex == 0 {
   lb.maxWeight -= lb.gcd
   if lb.maxWeight <= 0 {
    lb.maxWeight = maxWeight(lb.servers)
    if lb.maxWeight == 0 {
     return "" // No healthy servers
    }
   }
  }

  if lb.servers[lb.currIndex].Weight >= lb.maxWeight {
   return lb.servers[lb.currIndex].Name
  }
 }
}

func maxWeight(servers []Server) int {
 max := 0
 for _, server := range servers {
  if server.Weight > max {
   max = server.Weight
  }
 }
 return max
}

func main() {
 servers := []Server{
  {Name: "Server1", Weight: 4},
  {Name: "Server2", Weight: 2},
  {Name: "Server3", Weight: 1},
 }

 lb := NewLoadBalancer(servers)

 for i := 0; i < 10; i++ {
  server := lb.NextServer()
  if server != "" {
   fmt.Printf("Request %d routed to %s\n", i+1, server)
  } else {
   fmt.Printf("No healthy servers\n")
  }
```

```
 }
}
```

Least connections

```
package main

import (
 "fmt"
 "sync"
)

type Server struct {
 Name         string
 ActiveConns  int
 MaxConns     int
 ConnectionMu sync.Mutex
}

type LoadBalancer struct {
 servers []Server
}

func NewLoadBalancer(servers []Server) *LoadBalancer {
 return &LoadBalancer{
  servers: servers,
 }
}

func (lb *LoadBalancer) SelectServer() *Server {
 var selectedServer *Server
 minConns := -1

 for i := range lb.servers {
  server := &lb.servers[i]

  // Lock to safely access active connection count
  server.ConnectionMu.Lock()
  if minConns == -1 || server.ActiveConns < minConns {
   selectedServer = server
   minConns = server.ActiveConns
  }
  server.ConnectionMu.Unlock()
 }

 return selectedServer
}

func (lb *LoadBalancer) ServeRequest() {
```

```
 server := lb.SelectServer()

 if server != nil {
  // Simulate processing by incrementing active connections
  server.ConnectionMu.Lock()
  server.ActiveConns++
  server.ConnectionMu.Unlock()

  fmt.Printf("Request served by %s\n", server.Name)

  // Simulate request processing
  // ... Your request processing logic ...

  // Decrement active connections after request processing
  server.ConnectionMu.Lock()
  server.ActiveConns--
  server.ConnectionMu.Unlock()
 }
}

func main() {
 servers := []Server{
  {Name: "Server1", MaxConns: 10},
  {Name: "Server2", MaxConns: 10},
  {Name: "Server3", MaxConns: 10},
 }

 lb := NewLoadBalancer(servers)

 // Simulate serving requests
 for i := 0; i < 20; i++ {
  go lb.ServeRequest()
 }

 // Sleep to allow time for requests to complete
 // In a real application, you would use proper synchronization techniques
 select {}
}
```

Least response time

```
package main

import (
 "fmt"
 "math/rand"
 "sync"
 "time"
)
```

```
type Server struct {
 Name           string
 ResponseTimeMs int
}

type LoadBalancer struct {
 servers     []Server
 responseMu  sync.Mutex
 responseMap map[string]int
}

func NewLoadBalancer(servers []Server) *LoadBalancer {
 lb := &LoadBalancer{
  servers:     servers,
  responseMap: make(map[string]int),
 }

 // Initialize response time map
 for _, server := range lb.servers {
  lb.responseMap[server.Name] = 0
 }

 return lb
}

func (lb *LoadBalancer) SelectServer() string {
 minResponseTime := -1
 var selectedServer string

 // Find the server with the least response time
 lb.responseMu.Lock()
 for _, server := range lb.servers {
  responseTime := lb.responseMap[server.Name]
  if minResponseTime == -1 || responseTime < minResponseTime {
   selectedServer = server.Name
   minResponseTime = responseTime
  }
 }
 lb.responseMu.Unlock()

 return selectedServer
}

func (lb *LoadBalancer) ServeRequest() {
 serverName := lb.SelectServer()

 if serverName != "" {
  // Simulate request processing with random response time
  responseTime := rand.Intn(100) + 10

  fmt.Printf("Request served by %s (Response Time: %d ms)\n", serverName, respo
```

```
    // Update response time for the selected server
    lb.responseMu.Lock()
    lb.responseMap[serverName] += responseTime
    lb.responseMu.Unlock()
  }
}

func main() {
  servers := []Server{
    {Name: "Server1"},
    {Name: "Server2"},
    {Name: "Server3"},
  }

  lb := NewLoadBalancer(servers)

  // Simulate serving requests
  rand.Seed(time.Now().Unix())
  for i := 0; i < 20; i++ {
    go lb.ServeRequest()
  }

  // Sleep to allow time for requests to complete
  // In a real application, you would use proper synchronization techniques
  select {}
}
```

Load Balancer | Distributed Systems | Software Architect | Software Architecture

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience